Interface vs Abstract Class - Complete Comparison

You're Right! Both are Blueprints

Both **interfaces** and **abstract classes** serve as blueprints that define what methods subclasses must implement. However, they differ in **how much** and **what kind** of blueprint they provide.

Key Differences Summary

Feature	Interface	Abstract Class
Keyword	`interface`	`abstract class`
Implementation	`implements`	`extends`
Methods	All abstract (except default/static)	Can have both abstract and concrete
Variables	Only `public static final`	Any type of variables
Constructor	Cannot have	Can have constructors
Multiple Inheritance	Yes (implements multiple)	No (extends only one)
Access Modifiers	Methods implicitly public	Any access modifier
State	No instance variables	Can have instance variables
When to Use	"CAN-DO" relationship	"IS-A" relationship

Detailed Comparison with Examples

1. Interface - Pure Contract/Blueprint

java

```java
// Interface - Pure blueprint (contract)
interface Flyable {
    // All methods implicitly public abstract
    void fly();
    void takeOff();
    void land();

    // Variables are implicitly public static final (constants)
    int MAX_ALTITUDE = 50000;    // public static final
    String FLIGHT_STATUS = "AIRBORNE";

    // Default method (Java 8+) - provides implementation
    default void displayFlightInfo() {
        System.out.println("Max altitude: " + MAX_ALTITUDE + " feet");
    }

    // Static method (Java 8+)
    static void showFlightRules() {
        System.out.println("Follow aviation safety rules");
    }
}

interface Swimmable {
    void swim();
    void dive();

    default void floatOnWater() {
        System.out.println("Floating on water surface");
    }
}

// A class can implement multiple interfaces (multiple blueprints)
class Duck implements Flyable, Swimmable {
    private String name;

    public Duck(String name) {
        this.name = name;
    }

    // Must implement all abstract methods from all interfaces
    @Override
    public void fly() {
        System.out.println(name + " duck is flying");
    }

    @Override
```

```java
    public void takeOff() {
        System.out.println(name + " duck is taking off from water");
    }

    @Override
    public void land() {
        System.out.println(name + " duck is landing on water");
    }

    @Override
    public void swim() {
        System.out.println(name + " duck is swimming");
    }

    @Override
    public void dive() {
        System.out.println(name + " duck is diving for food");
    }
}
```

2. Abstract Class - Partial Implementation Blueprint

java

```java
// Abstract class - Partial blueprint with shared implementation
abstract class Animal {
    // Instance variables (state) - interfaces cannot have these
    protected String name;
    protected int age;
    protected String habitat;

    // Constructor - interfaces cannot have constructors
    public Animal(String name, int age, String habitat) {
        this.name = name;
        this.age = age;
        this.habitat = habitat;
        System.out.println("Animal " + name + " created");
    }

    // Concrete methods (shared implementation)
    public void sleep() {
        System.out.println(name + " is sleeping in " + habitat);
    }

    public void eat() {
        System.out.println(name + " is eating");
    }

    // Getters and setters
    public String getName() { return name; }
    public int getAge() { return age; }
    public String getHabitat() { return habitat; }

    // Abstract methods (must be implemented by subclasses)
    public abstract void makeSound();
    public abstract void move();
    public abstract String getSpecies();

    // Concrete method that uses abstract methods
    public void performDailyRoutine() {
        System.out.println("=== Daily Routine for " + name + " ===");
        eat();
        makeSound();
        move();
        sleep();
        System.out.println("Species: " + getSpecies());
    }

    // Protected method for subclasses
    protected void breathe() {
```

```
        System.out.println(name + " is breathing");
    }
}

// Single inheritance - can only extend one abstract class
class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age, "House");    // Call parent constructor
        this.breed = breed;
    }

    // Must implement all abstract methods
    @Override
    public void makeSound() {
        System.out.println(name + " barks: Woof! Woof!");
    }

    @Override
    public void move() {
        System.out.println(name + " runs on four legs");
    }

    @Override
    public String getSpecies() {
        return "Canis familiaris (" + breed + ")";
    }

    // Dog-specific methods
    public void wagTail() {
        System.out.println(name + " is wagging tail happily");
    }

    public String getBreed() {
        return breed;
    }
}
```

Multiple Inheritance vs Single Inheritance

Interface - Multiple Inheritance Allowed

java

```java
interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

interface CanWalk {
    void walk();
}

// Can implement multiple interfaces
class SuperDuck implements CanFly, CanSwim, CanWalk {
    @Override
    public void fly() { System.out.println("Flying high"); }

    @Override
    public void swim() { System.out.println("Swimming gracefully"); }

    @Override
    public void walk() { System.out.println("Walking on land"); }
}
```

Abstract Class - Single Inheritance Only

```java
```

```java
abstract class LandAnimal {
    protected int legs;
    public abstract void walk();
}

abstract class WaterAnimal {
    protected boolean hasGills;
    public abstract void swim();
}

// ❌ Cannot extend multiple abstract classes
// class Amphibian extends LandAnimal, WaterAnimal { } // Compilation Error

// ✅ Can only extend one abstract class
class Frog extends LandAnimal {
    public Frog() {
        this.legs = 4;
    }

    @Override
    public void walk() {
        System.out.println("Frog hops with " + legs + " legs");
    }
}
```

When to Use Which?

Use Interface When:

1. **Multiple inheritance needed** - class needs multiple contracts
2. **Pure contract** - just defining what methods must exist
3. **CAN-DO relationship** - "can fly", "can swim", "can draw"
4. **Unrelated classes** need same functionality
5. **Plugin architecture** - different implementations of same contract

Use Abstract Class When:

1. **Shared implementation** - common code for subclasses
2. **IS-A relationship** - "is an animal", "is a vehicle"
3. **Instance variables needed** - maintaining state
4. **Constructors required** - initialization logic
5. **Related classes** in same hierarchy

Complete Real-World Example

java

```java
// Interface for capabilities (CAN-DO relationship)
interface Drawable {
    void draw();
    void resize(double factor);

    default void display() {
        System.out.println("Displaying drawable object");
        draw();
    }
}

interface Colorable {
    void setColor(String color);
    String getColor();
}

// Abstract class for shared behavior (IS-A relationship)
abstract class Shape {
    protected double x, y;        // Position
    protected String name;        // Shape name

    // Constructor
    public Shape(String name, double x, double y) {
        this.name = name;
        this.x = x;
        this.y = y;
    }

    // Concrete methods (shared implementation)
    public void moveTo(double newX, double newY) {
        this.x = newX;
        this.y = newY;
        System.out.println(name + " moved to (" + x + ", " + y + ")");
    }

    public void showPosition() {
        System.out.println(name + " is at position (" + x + ", " + y + ")");
    }

    // Abstract methods (must be implemented)
    public abstract double calculateArea();
    public abstract double calculatePerimeter();

    // Getters
    public String getName() { return name; }
    public double getX() { return x; }
```

```
    public double getY() { return y; }
}

// Concrete class implementing interface and extending abstract class
class Circle extends Shape implements Drawable, Colorable {
    private double radius;
    private String color;

    public Circle(double x, double y, double radius) {
        super("Circle", x, y);    // Call abstract class constructor
        this.radius = radius;
        this.color = "Black";     // Default color
    }

    // Implement abstract methods from Shape
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }

    // Implement Drawable interface
    @Override
    public void draw() {
        System.out.println("Drawing " + color + " circle with radius " + radius);
    }

    @Override
    public void resize(double factor) {
        radius *= factor;
        System.out.println("Circle resized. New radius: " + radius);
    }

    // Implement Colorable interface
    @Override
    public void setColor(String color) {
        this.color = color;
        System.out.println("Circle color changed to " + color);
    }

    @Override
    public String getColor() {
        return color;
```

```
    }

    public double getRadius() {
        return radius;
    }
}

class Rectangle extends Shape implements Drawable, Colorable {
    private double width, height;
    private String color;

    public Rectangle(double x, double y, double width, double height) {
        super("Rectangle", x, y);
        this.width = width;
        this.height = height;
        this.color = "Blue";
    }

    // Implement abstract methods
    @Override
    public double calculateArea() {
        return width * height;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * (width + height);
    }

    // Implement interfaces
    @Override
    public void draw() {
        System.out.println("Drawing " + color + " rectangle " + width + "x" + height);
    }

    @Override
    public void resize(double factor) {
        width *= factor;
        height *= factor;
        System.out.println("Rectangle resized to " + width + "x" + height);
    }

    @Override
    public void setColor(String color) {
        this.color = color;
    }
```

```
    @Override
    public String getColor() {
        return color;
    }
}
```

Testing the Complete Example

java

```
public class BlueprintDemo {
    public static void main(String[] args) {
        System.out.println("=== CREATING SHAPES ===");
        Circle circle = new Circle(10, 20, 5);
        Rectangle rectangle = new Rectangle(0, 0, 8, 6);

        System.out.println("\n=== USING ABSTRACT CLASS METHODS ===");
        // Methods from abstract class
        circle.showPosition();
        circle.moveTo(15, 25);
        System.out.println("Circle area: " + circle.calculateArea());
        System.out.println("Circle perimeter: " + circle.calculatePerimeter());

        rectangle.showPosition();
        System.out.println("Rectangle area: " + rectangle.calculateArea());

        System.out.println("\n=== USING INTERFACE METHODS ===");
        // Methods from interfaces
        circle.setColor("Red");
        circle.draw();
        circle.resize(2.0);
        circle.display();   // Default method from Drawable

        rectangle.setColor("Green");
        rectangle.draw();
        rectangle.resize(0.5);

        System.out.println("\n=== POLYMORPHISM ===");
        // Abstract class polymorphism
        Shape[] shapes = {circle, rectangle};
        for (Shape shape : shapes) {
            System.out.println(shape.getName() + " area: " + shape.calculateArea());
        }

        // Interface polymorphism
        Drawable[] drawables = {circle, rectangle};
        for (Drawable drawable : drawables) {
            drawable.draw();
        }

        Colorable[] colorables = {circle, rectangle};
        for (Colorable colorable : colorables) {
            System.out.println("Color: " + colorable.getColor());
        }
```

```
    }
}
```

Evolution of Java Interfaces

Traditional Interfaces (Before Java 8)

java

```java
interface OldInterface {
    void method1();      // Only abstract methods
    void method2();      // All implicitly public abstract
    int CONSTANT = 100;  // Only constants allowed
}
```

Modern Interfaces (Java 8+)

java

```java
interface ModernInterface {
    // Abstract methods (traditional)
    void abstractMethod();

    // Default methods (have implementation)
    default void defaultMethod() {
        System.out.println("Default implementation");
    }

    // Static methods
    static void staticMethod() {
        System.out.println("Static method in interface");
    }

    // Private helper methods (Java 9+)
    private void helperMethod() {
        System.out.println("Private helper");
    }
}
```

Decision Tree: Interface vs Abstract Class

```
Do you need multiple inheritance?
├── YES → Use Interface
└── NO → Do you have shared implementation/state?
    ├── YES → Use Abstract Class
    └── NO → Do you need just a contract?
```

```
├── YES → Use Interface
└── NO → Use Regular Class
```

Common Patterns

1. Interface + Abstract Class Combination

java

```java
interface Processor {
    void process(String data);
}

abstract class BaseProcessor implements Processor {
    protected String name;

    public BaseProcessor(String name) {
        this.name = name;
    }

    protected void log(String message) {
        System.out.println(name + ": " + message);
    }
}

class DataProcessor extends BaseProcessor {
    public DataProcessor() {
        super("DataProcessor");
    }

    @Override
    public void process(String data) {
        log("Processing: " + data);
    }
}
```

2. Multiple Interfaces

java

```
interface Readable {
    String read();
}

interface Writable {
    void write(String data);
}

class File implements Readable, Writable {
    private String content = "";

    @Override
    public String read() {
        return content;
    }

    @Override
    public void write(String data) {
        content = data;
    }
}
```

Key Takeaways

Interfaces (Pure Blueprints):

- **What**: Contract defining what methods must exist
- **Why**: Multiple inheritance, pure abstraction, unrelated classes
- **When**: CAN-DO relationships, plugin architectures, contracts

Abstract Classes (Partial Blueprints):

- **What**: Base class with shared code and some abstract methods
- **Why**: Code reuse, shared state, IS-A relationships
- **When**: Related classes, shared implementation, constructors needed

Both Are Blueprints Because:

1. **Define structure** - specify what methods must exist
2. **Enforce implementation** - subclasses must implement abstract methods
3. **Enable polymorphism** - treat different objects uniformly
4. **Provide templates** - guide subclass development

The main difference is **how much blueprint** they provide:

- **Interface**: Minimal blueprint (just method signatures)
- **Abstract Class**: Rich blueprint (methods + implementation + state)

Both are essential tools for creating well-designed, maintainable object-oriented systems!